






**CODE GENERATION FOR BYTECODE COMPILER**

**Patent number:** JP2000040007  
**Publication date:** 2000-02-08  
**Inventor:** SHAYLOR NICHOLAS  
**Applicant:** SUN MICROSYST INC  
**Classification:**  
- international: G06F9/45; G06F9/44  
- european:  
**Application number:** JP19990184857 19990630  
**Priority number(s):**

Also published as:

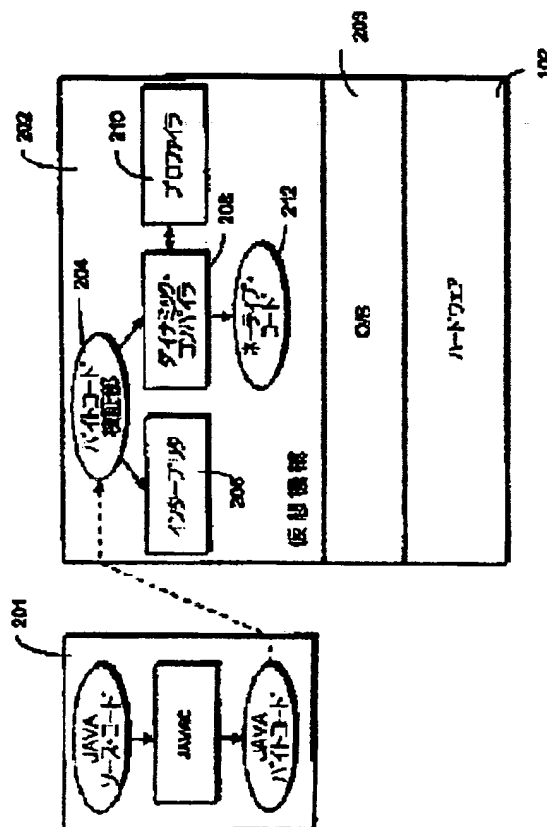
 US6760907 (B2)  
 US2002104076 (A1)  
 GB2342473 (A)  
 DE19928980 (A1)  
 CA2274210 (A1)

Report a data error here

**Abstract of JP2000040007**

**PROBLEM TO BE SOLVED:** To produce a native code in an execution time compiler from a bytecode group presented on a compiler and to provide a system which optimizes it.

**SOLUTION:** A compiler 208 accesses information 210 showing the possibility that a class is a specified type when an active program accesses. The compiler selects one code generation method among plural code generation methods by using the accessed information. A code generator produces an optimized native code in accordance with the selected code generation method and stores the optimized native code in a code cache 212 for reuse.



(19)日本国特許庁 (J P)

(12) 公 開 特 許 公 報 (A)

(11)特許出願公開番号

特開2000-40007

(P2000-40007A)

(43)公開日 平成12年2月8日(2000.2.8)

(51)Int.Cl. <sup>7</sup>	識別記号	F I	テーマコード(参考)
G 0 6 F 9/45		G 0 6 F 9/44	3 2 2 F
9/44	5 3 0		5 3 0 P

審査請求 未請求 請求項の数20 O L (全 13 頁)

(21)出願番号	特願平11-184857	(71)出願人	597004720 サン・マイクロシステムズ・インコーポレ ーテッド Sun Microsystems, In c. アメリカ合衆国カリフォルニア州94303, パロ・アルト, サン・アントニオ・ロード 901, エムエス・ピーエイエル01-521
(22)出願日	平成11年6月30日(1999.6.30)	(72)発明者	ニコラス・シェイラー アメリカ合衆国カリフォルニア州94560, ニューアーク, ポトレロ・ドライブ 6167
(31)優先権主張番号	09/108061	(74)代理人	100089705 弁理士 社本 一夫 (外4名)
(32)優先日	平成10年6月30日(1998.6.30)		
(33)優先権主張国	米国 (US)		

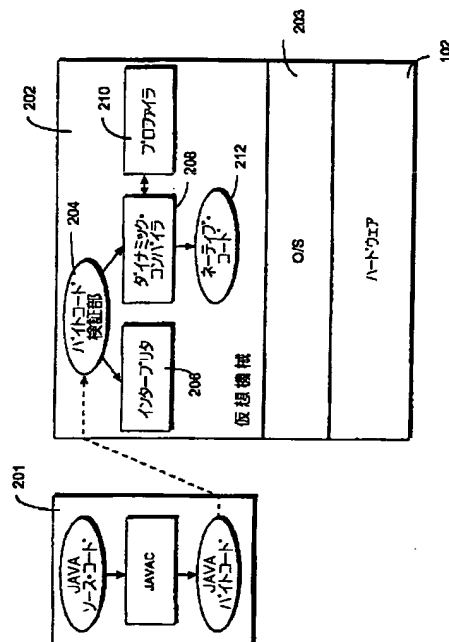
最終頁に続く

(54)【発明の名称】 バイトコード・コンパイラのためのコード生成

(57)【要約】

【課題】 コンパイラに提示したバイトコード群から、実行時コンパイラにおいてネイティブ・コードを生成し最適化するシステムを提供する。

【解決手段】 コンパイラ208は、実行中のプログラムがアクセスしたときに、クラスが特定のタイプである可能性を示す情報210にアクセスする。アクセスした情報を用いて、コンパイラは、複数のコード生成メソッドから、1つのコード生成メソッドを選択する。コード生成器は、選択したコード生成メソッドに応じて、最適化したネイティブ・コードを生成し、再利用のために、最適化したネイティブ・コードをコード・キャッシュ212に格納する。



【特許請求の範囲】

【請求項1】 バイトコードとして表現したプログラムにコードを生成するプロセスであって、実行すべきメソッドを表す1群のバイトコードを受け取るステップと、

受け取ったバイトコードが、他のメソッドをコールするコール側メソッドを表す場合、ターゲット・メソッドのクラスが明白に認められるか否かについて判定を行うステップと、

前記ターゲット・メソッドのクラスが明白に認められる場合、前記ターゲット・メソッドからのコードを、実行すべきコール側メソッドのコードにインラインするコードを、前記受け取ったバイトコードのために生成するステップと、

前記コール側メソッドの後の実行のために、前記生成したコードをセーブするステップと、から成るプロセス。

【請求項2】 請求項1記載のプロセスにおいて、前記ターゲット・メソッドのクラスが曖昧である場合、前記メソッドが、更に、

前記ターゲット・メソッドのクラスが、サブクラス化することが認められるものか否かについて判定を行うステップと、

前記ターゲット・メソッドのクラスが、サブクラス化することが認められる場合、前記メソッドに対する仮想関数コールを実施するコードを、前記受け取ったバイトコードのために生成するステップと、を含むこと、を特徴とするプロセス。

【請求項3】 請求項1記載のプロセスにおいて、前記ターゲット・メソッドのクラスが曖昧である場合、前記メソッドが、更に、

前記ターゲット・メソッドのクラスが、サブクラス化することが認められるものか否かについて判定を行うステップと、

前記ターゲット・メソッドのクラスが、サブクラス化することが認められる場合、前記ターゲット・メソッドのクラスに可能な全てのサブクラスが認められるか否かについて判定を行うステップと、

前記ターゲット・メソッドのクラスに可能な全てのサブクラスが認められる場合、前記メソッドは、更に、可能な各サブクラス毎にコード断片を生成するステップであって、各コード断片が、関連するサブクラスが定義するように、前記ターゲット・メソッドからのコードをインラインするステップと、

各コード断片毎にケースを有するスイッチを実施するコードを、前記受け取ったバイトコードのために生成するステップと、

前記コール側メソッドを実行した時点における前記ターゲット・メソッドの現サブクラス状態に基づいて、前記スイッチを選択するコードを、前記受け取ったバイトコードに生成するステップと、を含むこと、を特徴とする

プロセス。

【請求項4】 請求項1記載のプロセスにおいて、前記ターゲット・メソッドのクラスが曖昧である場合、前記メソッドが、更に、

前記ターゲット・メソッドのクラスとして正しい可能性があるクラスの集合を識別するステップと、

前記コール側クラスのあらゆる実行について、選択したクラスが正しい可能性が、正しくない可能性よりも高いという知識に基づいて、前記可能なクラス集合から1つのクラスを選択するステップと、

前記選択したクラスからのコードを、実行すべき前記コール側メソッドのコードにインラインする、第1コード断片を生成するステップと、

コードをインラインせずに、仮想関数コールを実施する第2のコード断片を生成するステップと、

前記第1および第2のコード断片の前に、前記コール側メソッドの各実行毎に、選択したクラスが正しいクラスか否かについて検査するように動作する検査コードを生成するステップと、

前記検査コードに応答して、前記第1のコード断片または前記第2のコード断片にいずれかを、コード実行に選択させる分岐コードを生成するステップと、

前記第1コード断片、第2コード断片、検査コードおよび分岐コードを結合し、前記受け取ったバイトコードのために生成した前記コードを形成するステップと、を含むこと、を特徴とするプロセス。

【請求項5】 請求項1記載のプロセスにおいて、前記ターゲット・メソッドのクラスが曖昧である場合、前記メソッドが、更に、

前記ターゲット・メソッドのクラスを固定化したかのように、前記ターゲット・メソッドのクラスからコードをインラインする第1のコード断片を、前記受け取ったバイトコードのために生成するステップと、

前記ターゲット命令に対する仮想関数コールを実施し、プログラムの実行を前記第1のコード断片に戻す、第2のコード断片を生成するステップと、

前記ターゲット・メソッドに対するエントリを備えたデータ構造を作成し、前記エントリに、前記第1のコード断片における第1の命令の第1のアドレスと、前記第2のコード断片の第1の命令の第2のアドレスを格納するステップと、

前記ターゲット・メソッドを無効化する毎に、前記第1のコード断片の前記第1の命令に、プログラムの実行を前記第2のコード断片に転送する第3のコード断片をバッチするステップと、を含むこと、を特徴とするプロセス。

【請求項6】 請求項5記載のプロセスにおいて、前記第3のコード断片が分岐命令を備え、前記第2のコード断片が、プログラムの実行を前記第1コード断片の終端に戻す分岐命令を含むこと、を特徴とするプロセス。

【請求項7】 請求項5記載のプロセスにおいて、前記第3のコード断片がコール命令を備え、前記パッチングするステップが、更に、前記第1のコード断片における第2の命令を、プログラムの実行を前記第1のコード断片の終端に戻す分岐命令と置換するステップを含むこと、を特徴とするプロセス。

【請求項8】 請求項5記載のプロセスにおいて、前記第3のコード断片が、ブレークポイント・ハンドラ・ルーチン呼び出す、ブレークポイント命令を備えること、を特徴とするプロセス。

【請求項9】 請求項8記載のプロセスにおいて、前記ブレークポイント・ハンドラ・ルーチンが、プログラムの実行を前記第1のコード・フラグメントの終端に戻すコードを含むこと、を特徴とするプロセス。

【請求項10】 請求項1記載のプロセスにおいて、前記ターゲット・メソッドのクラスがサブクラス化することが認められるものか否かについて判定を行う前記ステップが、実行すべき前記命令の以前の実行の履歴を試験するステップを含むこと、を特徴とするプロセス。

【請求項11】 受け取ったバイトコードからコードを生成する装置であって、

前記バイトコードを受け取るコンパイラと、

前記コンパイラ内にあり、複数のエントリを有するデータ構造であって、各エントリが、前記バイトコードの一意の群によって参照されるオブジェクト指向プログラム・クラスに対応する、データ構造と、

前記データ構造の各エントリ内にあり、前記対応するクラスが、サブクラス化する可能性が高いか否かを示すデータ・レコードと、

前記コンパイラ内にあり、前記データ・レコードにアクセスするように結合したコード生成器であって、各バイトコード群毎に、当該バイトコード群が参照するクラスに対応する前記データ・レコードの現在値に基づいて、複数のコード生成オプションの中から選択するコード生成器と、

複数のエントリを有するネイティブ・コード・キャッシュであって、各エントリが特定のバイトコード群に対応し、各エントリが、前記特定のバイトコード群と関連付けて生成したネイティブ・コードを保持し、前記コンパイラが特定のバイトコード群を最初に受け取る毎に、前記バイトコード群を前記コード生成器に渡し、前記コンパイラが前記特定のバイトコード群を後に受け取る毎に、前記ネイティブ・コード・キャッシュから選択したコードに、前記バイトコードを変換する、ネイティブ・コード・キャッシュと、から成る装置。

【請求項12】 請求項11記載の装置であって、更に、ある時間期間にわたって前記コンパイラを監視するように結合してあり、プログラムの実行の間に呼び出された実際のクラスおよびサブクラスに関する知識を蓄積する

プロファイラを備えること、を特徴とする装置。

【請求項13】 オブジェクト指向プログラム・コードの実行時コンパイル・システムであって、前記プログラム・コードが、複数のオブジェクト指向クラスを定義するコードを含み、

前記プログラム・コードを受け取り、ターゲット・メソッドをコールするコール側メソッドを識別し、前記ターゲット・メソッドのクラスが明確に認められるか否かについて判定を行う実行時コンパイラと、

前記コンパイラ内にあり、前記ターゲット・メソッドからのコードを、実行すべきコール側メソッドのコードにインラインするコード生成器と、

前記コール側メソッドのその後の実行のために、生成したコードを格納するデータ構造を備えたコード・キャッシュと、を備えること、を特徴とするシステム。

【請求項14】 コンピュータ・システムであって、バイトコード・プログラムを格納するメモリであって、該バイトコード・プログラムが一連のバイトコードを含み、複数のバイトコード群が、コール側メソッドおよびターゲット・メソッドを含む、複数の実行すべきメソッドを表し、前記コール側メソッドが、前記ターゲット・メソッドをコールする動作を含む、メモリと、

ネイティブ・コードを実行するデータ処理ユニットと、前記メソッドの少なくともいくつかのネイティブ・コード表現を格納するコード・キャッシュと、

特定のバイトコード群を最初に受け取ったときに、当該受け取ったバイトコード群を、前記バイトコードのネイティブ・コード表現に変換し、該ネイティブ・コード表現を前記コード・キャッシュに格納するように動作するコンパイラと、

前記データ処理ユニットに結合したプロファイラであって、各メソッドのクラスについて、当該メソッドのクラスがサブクラス化する可能性を示す情報を格納するデータ構造を有する、プロファイラと、

前記コンパイラ内にあり、前記プロファイラのデータ構造に結合してあり、前記格納した情報に基づいて、コード生成メソッドを選択する、最適化コードと、から成るコンピュータ・システム。

【請求項15】 請求項14記載のコンピュータ・システムにおいて、前記プロファイラが、更に、各メソッドの実行を監視し、各メソッドのクラスがサブクラス化するときを記録するモニタを備え、

前記コンピュータ・システムが、更に、前記プロファイラのデータ構造内に格納した情報を変更し、前記記録したモニタ情報を示す履歴アキュムレータを備えること、を特徴とするコンピュータ・システム。

【請求項16】 請求項14記載のコンピュータ・システムにおいて、前記最適化コードが、更に、プロファイラのデータ構造に結合してあり、メソッドのクラスが明確に認められる場合を識別するコードと、

プロファイラのデータ構造に結合してあり、メソッドのクラスが動的なタイプであるが、特定のタイプである可能性の方がそうでない可能性よりも高い場合を識別するコードと、

プロファイラのデータ構造に結合してあり、メソッドのクラスが動的であるが、いずれの記録した実行においてもサブクラス化されたことが全くない場合を示すコードと、

プロファイラのデータ構造に結合してあり、メソッドのクラスが動的であり、いずれの特定の実行時においても、複数のタイプのいずれか1つである可能性が高い場合を識別するコードと、を備えること、を特徴とするコンピュータ・システム。

【請求項17】 請求項16記載のコンピュータ・システムにおいて、前記コンパイラが、更に、コードが明確に認められるという識別にตอบสนองし、いずれかのコール側メソッドのためにコードを生成する場合には、前記メソッドを表すネーティブ・コードを、前記コール側メソッドのネーティブ・コードにインラインするコンパイル方法を備えること、を特徴とするコンピュータ・システム。

【請求項18】 請求項16記載のコンピュータ・システムにおいて、前記コンパイラが、更に、メソッドのクラスが動的なタイプであるが、特定のタイプのものである可能性の方がそうでない可能性よりも高いという識別にตอบสนองし、

正しい可能性の方が正しくない可能性よりも高いと識別された前記特定のタイプの前記クラスから前記メソッドを識別するネーティブ・コードをインラインし、前記メソッドのクラスの現タイプに対する仮想関数コールを実施し、

正しい可能性の方が正しくない可能性よりも高いと識別された前記特定のタイプの前記クラスが実際には正しいか否かを検査し、

前記検査に基づいて、前記インライン・ネーティブ・コードまたは前記仮想関数コール・コードのいずれかに分岐する、コードを生成するコンパイル方法を備えること、を特徴とするコンパイル・システム。

【請求項19】 請求項16記載のコンピュータ・システムにおいて、前記コンパイラが、更に、メソッドのクラスが動的であるが、いずれの記録した実行においてもサブクラス化されたことが全くないという識別にตอบสนองし、前記ターゲット・メソッドのクラスから前記ターゲット・メソッドを表すネーティブ・コードをインラインする第1のコード断片と、前記ターゲット・メソッドのクラスに対する仮想関数コールを実施する第2のコード断片とを生成するコンパイル方法と、前記第1のコード断片を保持する、前記コード・キャッシュ内の第1の位置と、前記第2のコード断片を保持する、前記コード・キャッ

シュ内の第2の位置と、

前記第1の位置のアドレスと前記第2の位置のアドレスとを保持するデータ構造と、

前記ターゲット・メソッドのクラスがサブクラス化したか否かについて検出するコードと、

前記ターゲット・メソッドのクラスがサブクラス化したことの検出にตอบสนองし、前記コード・キャッシュ内の前記第1の位置のアドレスに、ブレークポイント命令をパッチするコードと、

前記コード・キャッシュ内の前記第2の位置にあるコードを実行し、前記コード・キャッシュ内の前記第1の位置の終端のアドレスに戻る、ブレークポイント処理ルーチンと、を備えること、を特徴とするコンピュータ・システム。

【請求項20】 コンピュータに結合する搬送波に具体化し、バイトコードとして表すプログラムのために、前記コンピュータにおいてコードを生成するコンピュータ・データ信号であって、

前記コンピュータに、実行すべき方法を表すバイトコード群を受け取らせるように構成したコードから成る第1のコード部分と、

前記コンピュータに、前記ターゲット・メソッドの各実行毎に、前記ターゲット・メソッドのクラスが特定のタイプのものである可能性が高いか否かについて判定を行わせるように構成したコードから成る第2のコード部分と、

前記第2のコード部分に結合してあり、前記コンピュータに、前記第2のコード部分が行った決定に基づいて、コード生成計画を選択させるように構成したコードから成る第3のコード部分と、

前記第3のコード部分に結合してあり、前記選択したコード生成計画に応じて、前記バイトコード群を実装するコードを生成する第4のコード部分と、を備えるコンピュータ・データ信号。

【発明の詳細な説明】

【0001】

【発明の属する技術分野】 本発明は、一般的に、コンパイラに関し、更に特定すれば、バイトコードとして表現したプログラムの実行を最適化するためのコード生成技法に関するものである。

【0002】

【従来の技術】 Java™プログラミング言語 (Sun Microsystems Inc. (サン・マイクロシステムズ社の商標) のようなバイトコード・プログラミング言語は、バイトコードの集合として、コンピュータ・プログラムを表現する。各バイトコードは、クライアント・コンピュータ上のソフトウェアにのみ存在する「仮想機械」のための数値機械コードである。仮想機械とは、本質的にはインタープリタであり、バイトコードを理解し、このバイトコードを機械コードに翻訳し、次いでネーティブ機

械 (native machine) 上でこの機械コードを実行する。

【0003】Javaプログラミング言語のようなバイトコード・プログラミング言語は、ソフトウェア・アプリケーション開発者の間で好評を勝ち得ている。何故なら、これらは、使用が容易でありしかも移植性が高いからである。バイトコードとして表現したプログラムは、バイトコードを適正に解釈し変換可能な仮想機械を有するコンピュータであれば、いずれにも容易に移植することができる。しかしながら、バイトコード・プログラミング言語は、実行時にクライアント・コンピュータ上で解釈しなければならないので、CまたはC++のような従来からのコンパイル型言語と勝負になる速度の実行は不可能であるという欠点があった。

【0004】バイトコード言語の速度の限界は、主に、コンパイル・プロセスに関係する。コンパイルは、高水準言語（即ち、人が読むことができる言語）で著したプログラムを機械読み取り可能コードに変換するプロセスのことである。コンパイル・プロセスには、4つの基本的な工程がある。即ち、トークン化 (tokenizing)、解析、コード生成、および最適化である。従来のコンパイル型プログラムでは、これらの工程は全て、実行時に完了する。これに対して、BASICのようなインタープリタ型言語では、コンパイル工程の全ては、実行時に各命令毎に行う。CSHのようなコマンド・シェル (command shell) も有限数のコマンドを認識するインタープリタの例である。インタープリタ型言語は、得られたコードを最適化する方法がないので、非効率となる。

【0005】バイトコード・プログラミング言語では、トークン化および解析は、実行時に行われる。解析後に、仮想機械が解釈できるバイトコードにプログラムを変換する。その結果、バイトコード・インタープリタは、元来のBASICプログラミング言語の実施のいくつかにおけるような言語インタープリタよりは高速となる。また、得られたプログラムは、バイトコード・フォーマットで表現した場合、完全にコンパイルしたプログラムよりも簡潔である。これらの特徴のために、バイトコード言語は、実行のためにソフトウェアをある機械から別の機械に転送することがあるネットワーク状コンピュータ環境では、有用な折衷案となった。

【0006】Javaプログラミング環境では、バイトコードへの変換を行うプログラムのことを「java c」と呼ぶが、Javaコンパイラと呼ぶ場合もある（しかし、これはコンパイル・プロセスの一部を行うに過ぎない）。クライアント・コンピュータ上でバイトコードを解釈するプログラムのことを、Java仮想機械 (JVM) と呼ぶ。他のインタープリタと同様、JVMは、それが受け取った各バイトコードを実行するループ内で走る。しかしながら、バイトコードを解釈するので、仮想機械上でもなお時間のかかる変換工程がある。各バイトコード毎に、インタープリタは、対応する一連

の機械命令を識別し、次いでこれらを実行する。いずれの変換に伴うオーバーヘッドも、1回では取るに足りないが、実行する命令毎にオーバーヘッドが蓄積する。大きなプログラムでは、オーバーヘッドは、完全にコンパイルした一連の命令を単純に実行する場合と比較すると、大量となる。その結果、Javaプログラミング言語で書いた大きなアプリケーションは、同等のアプリケーションを完全にコンパイルした形態よりは遅くなる傾向がある。

【0007】実行を高速化するために、仮想機械をジャスト・イン・タイム・コンパイラ即ちJIT (just-in-time) と結合したり、あるいはそれを内蔵している。JITは、バイトコードを実行する前に、これらをネイティブ機械コードにコンパイルすることによって、バイトコード・インタープリタの実行時パフォーマンスを向上させる。JITは、最初に一連のバイトコードを検出したときに、これらを機械命令に変換し、次いで機械命令を実行するので、バイトコードを順次呼び出して解釈することはない。機械命令は、メモリ内を除外と、どこにもセーブされないの、次回プログラムを走らせるとき、JITコンパイル・プロセスを新たに開始する。

【0008】その結果、バイトコードはなおも移植可能であり、しかも多くの場合、通常のインタープリタにおけるよりも、はるかに高速に走るようになる。ジャスト・イン・タイム・コンパイルは、特に、多くの計算プログラムのように、コード・セグメントを繰り返し実行する場合に有用である。また、ジャスト・イン・タイム・コンパイルは、パフォーマンスの改善には殆ど寄与せず、実際には、1回または数回実行した後再使用しない小さなコード部分では、パフォーマンスが遅くなる場合もある。

【0009】JIT技術の欠点の1つとして、コンパイルを実行時に行うので、機械コードを最適化しようとして費やす計算時間がオーバーヘッドとなり、プログラム実行の低速化を招く場合があることがあげられる。したがって、多くの最適化技法は、従来のJITコンパイラにおいては実用的でない。また、JITコンパイラは、一度に大量のコードを見ないので、大量のコードを最適化することができない。この結果、コンパイラは、プログラムが用いたクラス集合の確実な判断ができない。更に、クラス集合は所与のプログラムを実行する毎に変化する可能性があるの、仮想機械コンパイラは、クラス集合が常に明確に認められると想定することが全くできない。

【0010】この不確実性のため、実行時に真の最適ネイティブ・コードを生成することが困難となる。クラス集合の知識が不完全のまま最適化を試しても、非効率となる可能性があり、あるいはプログラムの機能性を変えてしまう虞れもある。クラス集合に関する不確実性は、Java実行の広い範囲にわたって潜在的な非効率

性を生ずる。クラス集合とは、Java言語プログラムの特定のインスタンスを実行するために用いる全てのクラスの集合のことである。従来からのバッチ・コンパイル型プログラムでは、コンパイル時にプログラムが用いるクラス全体がわかるので、最適化の作業は非常に簡単である。

【0011】共通プログラム発生 (common program occurrence) とは、「メソッド・コール」と呼ぶプロセスにおいて、第1のメソッド (コール元メソッド) が第2のメソッド (ターゲット・メソッド) をコールすることである。このメソッド・コール・プロセスは、プログラマの観点からは利点があるが、多くのクロック・サイクルを消費する。オブジェクト指向プログラム・コードのコンパイルにおいて重要な最適化に、「インライニング」 (inlining) と呼ぶものがある。完全にコンパイルしたプログラムでは、インライニングは、ターゲット・メソッドのコードをコール元メソッドにコピーすることによって、これらのターゲット・メソッドのコールを一層効率的に行う。

【0012】しかしながら、Javaプラットフォームの意味論 (semantics) のため、コンパイラはクラス集合全体を全く判断することができない。クラスは拡張可能であり、Javaに動的にロード可能であるので、後から発生するクラスの拡張が、メソッドを無効化してしまう可能性があり、コンパイラは、仮想メソッド・コールがいずれかの特定のメソッドに到達することを、確信を持って認めることができない。Javaプログラミング言語では、デフォルトのメソッドは、「最終」と明示的に定義していなければ、無効化される可能性がある。「非最終」メソッドは全て、無効化可能と見なされる。

【0013】したがって、非最終リーフ・メソッドに対する全てのコールは、ターゲット・メソッドが後のある時点で無効化される可能性があることを想定しなければならない。このため、以前はこれら呼び出すために、時間のかかる仮想メソッド・コール・シーケンスを用いてきた。メソッドは小さい程、インライニングは一層有利となる。典型的に、1ライン・メソッドは、その内容を実行するよりも、ルーチンの開始および終了にはるかに時間がかかる。メソッドにオーバーロードするクラスが今後ないことを確実に知ることができれば、多くの場合これら非最終リーフ・メソッド・コールの85%までもが解消可能であることが、検査によって明らかとなった。

【0014】

【発明が解決しようとする課題】プログラムに対するクラス集合が明確に認められない環境においても、できる限り最適なネイティブ・コードを生成する方法および装置が必要とされている。また、クラス集合の知識が不完全な場合に対処し、プログラムの実行パフォーマンスへの影響を許容できる程度にする方法および装置も必要と

されている。

【0015】

【課題を解決するための手段】端的に述べると、本発明は、実行時コンパイラにおいて、当該コンパイラに提示する1群のバイトコードからネイティブ・コードを生成し最適化する方法、システムおよび装置に関する。コンパイラは、実行中のプログラムがクラスにアクセスするときに、そのクラスが特定のタイプである尤度を示す情報にアクセスする。このアクセスした情報を用いて、コンパイラは複数のコード生成メソッドから1つのコード生成メソッドを選択する。コード生成器は、選択したコード生成メソッドに応じて、最適化したネイティブ・コードを生成し、再利用のために最適化ネイティブ・コードをコード・キャッシュに格納する。

【0016】

【発明の実施の形態】本発明は、同じバイトコード言語プログラムを繰り返し走らせるために用いるシステムが、「プロファイリング」 (profiling) と呼ぶプロセスによって、これらのプログラムにおいて用いたクラス集合について、適切かつ正確な知識を蓄えることができるという前提を基にしている。プロファイリングは、単一のプログラムの実行、または異なるセッションで行われる多数のプログラムの実行に対して行うことができる。従来システムの欠点、即ち、実行時における特定の知識の欠如を利点に変えるのは、実行時に得られるこの知識の蓄積である。この利点が得られるのは、蓄積した知識が特定の実行時環境に特化され、実行中のプログラムのプロファイリングを行うときに実行時環境のあらゆる特質 (idiosyncrasies) を本質的に考慮するからである。対照的に、従来のコンパイル・プロセスは、典型的な実行時環境における典型的なプログラムの挙動に対する一般化された知識に基づいて最適化を行っていた。

【0017】本発明によるシステムは、以前の実行において用いられたクラスを拡張することはないことを「認めた」とき、または理由があつてそう信じることができるときに、バイトコード・プログラムから効率的なネイティブ・コードを生成する。「認める (know)」および「知識 (knowledge)」という用語は、ここで用いる場合、システムおよび/またはソフトウェア内のデータ構造が、既知の情報を表すデータを保持していることを意味する。同様に、理由がある確信とは、システムおよび/またはソフトウェアが、ある状態を確実に認めることができなくても、その状態は存在しないよりも存在する確率の方が高いことを示す情報を保持していることを示す。

【0018】本発明は、実行中のプログラムについて認めた情報または理由があつて確信した情報に応答し、ダイナミック・コンパイラを用いて、実行時にコードを生成する。特定の例では、ダイナミック・コンパイラは、多数のコード生成メソッドを含み、コンパイルしている

いずれの特定のバイトコード群についても、その中から選択することができる。どのメソッドをそれぞれの場合に用いるかという選択は、コード自体を検査することによって瞬時的に得ることができる知識、およびコードの実行を監視し経時的に収集した知識に基づいて行う。

【0019】本発明は、プログラムにおける全てのバイトコードをネイティブ・コードにコンパイルするシステムに関して説明するが、本発明は、選択したバイトコード群のみをコンパイルするシステムに採用しても有用であることは理解されよう。例えば、本発明による仮想機械は、ダイナミック・コンパイラと並行に動作する、従来のバイトコード・インタプリタを含んでもよい。このような実現例では、選択したバイトコード群の実行頻度、これらのバイトコードをコンパイルすることによって達成し得る相対的な効果、またはその他のパフォーマンスに基づく評価基準を基に、これらをコンパイルすることができる。したがって、これらの実現例は、本発明の目的のためにここに記載する具体的な実現例と同等である。

【0020】具体的には、本発明は、コードがターゲット・メソッド・コールを伴う場合に、インライン化技法を駆使することによってネイティブ・コードの最適化を図ることを目的とする。したがって、本発明は、オブジェクト指向プログラムの最適化において、他のオブジェクトからメソッドやそれらのメソッドを記述するクラスを頻繁にコールする、多くの小さなオブジェクトやメソッドを利用する場合に、特に有用である。Javaプログラミング言語は、目的が限られた小さなメソッドを用いて厳格なオブジェクト定義を促進(encourage)する高度なオブジェクト指向言語の一例である。

【0021】高度なオブジェクト指向言語では、クラス拡張性が、プログラミングを容易にし、機能性を改善する重要な特性(key feature)となる。クラスの拡張とは、これがサブクラス化され、このクラスが記述したメソッドが無効化されるときのことである。一旦無効化されると、無効になったメソッドに対する後続のコールは全て、新しいメソッドを用いるが、このメソッドは、メソッド・コールの受け取り側が拡張クラスである場合、親クラス即ち抽象クラス(abstract class)に元々記述されているメソッドではない。本発明の重要な有用性は、実行中のシステムにおいてサブクラス化を行うことに起因する複雑性に対処するシステムおよび方法を提供することにある。

【0022】図1は、本発明による方法および装置を実施するように構成したコンピュータ・システム100を示す。クライアント・コンピュータ102は、サーバ・コンピュータ104上に位置するプログラムをダウンロードする。クライアント・コンピュータは、プログラム命令を実行するプロセッサ・ユニット106を有する。プロセッサ・ユニット106は、システム・バスを通じ

てユーザ・インターフェース108に結合してある。ユーザ・インターフェース108は、ユーザに情報を表示するために使用可能なデバイス(例えば、CRTまたはLCDディスプレイ)、およびユーザからの情報を受け入れるデバイス(例えば、キーボード、マウス等)を含む。メモリ・ユニット110(例えば、RAM、ROM、PROM等)は、プログラム実行のためのデータおよび命令を格納する。記憶装置112は、大容量記憶装置(例えば、ハード・ディスク、CDROM、ネットワーク・ドライブ等)から成る。モデム114は、システム・バスからのデータを、ネットワーク105を介した伝送に適したフォーマットに、あるいはその逆に変換する。また、モデム114は、ネットワーク・アダプタ、あるいは通信ネットワーク用のその他の純粋なデジタル・アダプタまたは混成アナログ・デジタル・アダプタと交換しても同等である。

【0023】サーバ104は、典型的に、同様のコンポーネント群から成り、プロセッサ116、ユーザ・インターフェース118、およびサーバ・メモリ120を含む。特定の例では、サーバ記憶装置122がバイトコードで表現したプログラムを格納し、このプログラムをモデム114を介しネットワーク105を通じてクライアント・コンピュータ100に送信する。尚、本発明は、図1に示すネットワーク・コンピュータ環境以外にも、単一のコンピュータ上でも実施可能であることは理解されよう。その場合、バイトコード・プログラムを単にクライアント記憶装置112に格納すればよい。

【0024】本発明の説明において、「クラス・タイプ」という用語は、特定のクラスを識別し、他のクラスまたは当該クラス自体のいずれのサブクラスとも、当該クラスを区別する特性のことを言う。クラスをサブクラス化した場合、新たなクラス・タイプが生ずる。特定のオブジェクトのクラス・タイプは、クラス記述に格納したデータから判定することができる。クラスは、「最終」キーワードを用いて定義することができるが、その場合クラスをサブクラス化することはできない。その他の場合、サブクラス化は可能である。オブジェクト指向プログラミングはサブクラス化を促進するので、大抵の場合(実際には、Javaプログラミング言語におけるデフォルトの場合)、クラスはサブクラス化可能である。クラスがサブクラス化可能である場合、何回でもサブクラス化することができ、その結果、プログラムの実行全体に対して、クラス・タイプの集合が得られる。実行時に、メソッド・コールによってあるメソッドをコールする場合、このタイプ集合の内正しいのは特定のタイプ1つのみである。

【0025】図2は、Javaプログラミング環境の一例を示し、「コンパイル時環境」201、および「実行時環境」202を含む。Java言語プログラムの開発および実行は、2つの工程を必要とする。プログラマ



は、Javaソース・コードをタイプ入力する。これを、図2に示すjavacのようなJavaコンパイラによって、Javaバイトコードと呼ぶ形態に変換する。前述のように、Javaバイトコードは、簡潔でありしかも移植可能であるので、ネットワーク・コンピュータ・システムにおいてプログラムを格納したり転送するには、理想的な形態である。

【0026】次に、バイトコードを実行時環境202に転送し、Java仮想機械(JVM)のような公知のプログラムによって実行する。全てのJVMは同じバイトコードを理解するので、Javaプログラムのバイトコード形態は、JVMを備えたあらゆるプラットフォーム上で走らせることができる。このように、JVMはJavaバイトコードのための汎用実行エンジンである。所与のプラットフォームのためにJVMを一旦書くことができれば、あらゆるバイトコード・プログラムをそれによって走らせることができる。従来のJVMにおけると同様、好適な実現例は、バイトコード・インタープリタ206を含み、ダイナミック・コンパイラ208と並列に走って、コンパイラ208が時間を費やして最適化したネイティブ・コードを生成している間に、従来と同様に、解釈したネイティブ・コードを与える。

【0027】本発明によるダイナミック・コンパイラ208は、前述のようにコンパイルを最適化することにより、Javaコードの実行を高速化する。コンパイラ208は、Javaバイトコードを取り込み、プロファイル210から得た知識およびパフォーマンス情報にตอบสนองして、ユーザのハードウェア102の具体的な種別およびオペレーティング・システム203のためのネイティブ・コードに、バイトコードを変換する。本発明は、コンパイラ技術の最適化を実施し、プログラムをネイティブに実行可能な形態に変換することによって、高いパフォーマンスを得る。インタープリタ206からの解釈したコードとは異なり、コンパイラ208からの最適化コードは、後に再利用するために、コード・キャッシュ212にセーブしておく。

【0028】プロファイル210は、経時的にコードの実行を監視し、主に、実行中の特定のプログラムのサブクラス化およびオーバーロード挙動を監視する。本発明の重要性として、プロファイル210は、プログラムが実際にどのクラスを使用しているかに関する知識を蓄積する。プロファイル210は、プロファイル・データを生成し、クラス毎またはメソッド毎に、プロファイル・データ構造(図示せず)に格納する。したがって、ダイナミック・コンパイラ208はクラス集合について先験的知識を全く有さないが、クラス集合およびクラス挙動に関してセッションに特化した知識を、プロファイル210から得ることができる。実現例の一例では、プロファイル210は、実行を監視する各クラス毎にエントリを有する、テーブルのようなデータ構造を含む。

【0029】実行が進むに連れて、プロファイル210は、データ構造にプロファイル・データを格納し、特定のクラスがサブクラス化されたか否かについて示すと共に、特定のクラスの1つのクラス・タイプが、いずれの特定メソッドの呼び出し時においても、正しいクラス・タイプである可能性が最も高いか否かについて示す。このように、格納したデータは、クラスの記述から直接判定可能な単純な「固定」指示よりも有用である。更に、プロファイル210内のデータ構造は、起動時に、何らかの既知状態に初期化することができるが、動的に変化して、これを用いている環境におけるプログラムの現挙動を反映している。

【0030】本発明は、メソッド・コールの内、本発明が特に有用な4つのカテゴリに関して理解すると役に立つ。第1のケースは、受け取り側のオブジェクトのクラス(即ち、ターゲット・メソッドを定義するクラス)が明確に認められる場合のメソッド・コールである。これは、メソッドが「最終」として示されたとき(即ち、その記述の最終というキーワードを含む場合)、または、明示的に最終と宣言されたのではないが、実際には当該メソッドが最終であることが固定コード分析によって解明されたときに、得ることができる。第2のケースでは、受信側のクラスが、可変であることが認められている。抽象クラスに定義してあるメソッド、またはサブクラス化することが認められているクラスへのコールが、このケースの例である。これら最初の2つのケースは、クラスが明確に固定であることが認められる場合、および特定の執行においてどのタイプとなるかについてクラスが非常に曖昧である場合という、極端なケースを表す。

【0031】残りの2つのケースは、クラスに関する何らかの執行時知識を利用して、より効率的なコードを生成可能な、不明確な領域(gray area)を含む。第3のケースでは、受け取り側のクラスは、ある特定のタイプである場合の方がそうでない場合よりも多いものとして識別する。本発明によれば、このケースは、プロファイルに基づくフィードバックによって識別することができる。第4のケースでは、受け取り側のクラスは、以前の執行において全くサブクラス化しなかったことを、プロファイルに基づくフィードバックが示したことにより、このクラスは特定のタイプであることの可能性が非常に高い。

【0032】本発明によれば、第1のケースでは、後続のメソッド・コールによって、クラスが無効化され得ないことがわかっている。この場合、コール側に対して生成したコードに、メソッド・コールを常に直接コールするか、またはインラインすることができる。これら2つのタイプの最適化(直接コールおよびインライン化)は、別個であるが、類似した動作である。直接コールは、仮想メソッド・コールよりも実行が速い。更に高速

化した最適化は、ターゲット・メソッドのコードをコール側メソッドのコードに実際にコピーすることである（即ち、インライニング）。しかしながら、インライニングは、実行時にコード・サイズの増大を招くため、必要なコンピュータの命令キャッシュが多くなる。特に指定しない限り、これら2つの最適化技法間の選択は、個々の用途の必要性を満たすのに適したコードの生成に係ることである。

【0033】ケース2では、本発明を単純に実施すると、従来の仮想機能コールを実施するコードを生成することになる。最適化はこのような場合にはより多くのオーバーヘッドを必要とし、その結果メソッドのコール毎に非常に大量のコードを生成する場合があるので、これはもっともなことである。あるいは、全ての可能なクラスを「スイッチ・ステートメント」(switch statement)における場合として識別し、各可能なクラスからのターゲット・メソッドのコードを、コール側メソッドに対するコードにインラインする。スイッチは、現クラス・タイプを検査し、検査結果に基づいてスイッチを選択することにより、コール元メソッドを実行するときに選択する。

【0034】第3のケースでは、コールは、ターゲット・メソッドのコードにおいて小さな検査を行うことによって、最も正しい可能性が高いメソッドに直接行うことができる。クラスがそのメソッドに対する正しいクラスでない場合、仮想関数コールを行うことができる。ある

class Y having the description:

```
class Y {
    int value;           //declare "value" as an
                        //data type
    integer              //data type
    int length() {       //declare length() as a
    method               return value; //that returns
    an integer "value"
    }                   //when called
}
```

【0038】例えば、コール側メソッドは、  
x=y.length();  
とすることができる。

【0039】このコール側メソッドのために生成したネ

いは、ケース3のメソッドは、インラインすることができ、インラインするコードの最初の命令の前に検査を行い、クラス・タイプが正しいか否かチェックし、正しくない場合代わりに仮想メソッド・コールを行う。

【0035】ケース4では、メソッドのコールは、メソッドを直接コールすることによって効率的に行うことができる。本発明によれば、生成するコードは、クラスが決してサブクラス化しないことを始めに想定する。しかしながら、後にこれがサブクラス化した場合、メソッドの最初の命令を小さなコード・フラグメントにパッチし(patch)、受信側を検査し、それが正しくない場合仮想関数コールを行う。それが正しい場合、パッチした命令を実行し、分岐して主メソッド・コードに戻る。

【0036】ターゲット・メソッドが短い程、インラインすることは一層重要となる。コール側メソッドの再コンパイルを回避するためには、受け取り側クラス（即ち、ターゲット・メソッドのクラス）の検査が必要となる。望ましくは、この検査はオーバーヘッドを殆ど追加せず、比較的素早いとよい。単一命令ターゲット・メソッドをコールするメソッドの最悪の場合の例を、単純なアクセッサ・メソッド(accessor method)によって例証する。例えば、以下の記述を有するクラスYを想定する。

【0037】

【表1】

ーティブ・コード・セグメントは、次のようになる。

【0040】

【表2】

```

cmp [y+class],#constantForClassY //compare y's
class type                          //to a
specified constant
bne vfCall                          //branch when
the
                                   //comparison
is not
                                   //equal
mov[y+value],x                     //move the
returned
                                   // "value" to x

```

【0041】上述機械コードでは、最初の命令は、メモリ内のクラス記述から読み出したクラス・タイプを、予期するクラスに対する所定の定数と比較する。この所定の定数は、コードを生成するときに決定し、コードを再生成するときはいつでも更新することができる。2つの値が等しい場合、等しくないときに分岐する命令（branch-on-not-equal instruction）を飛ばし、インライン・メソッドを実施する機械コードを実行する。仮想関数テーブルのコール・スルー（call-through）の場合は、vfCallにおいて処理し、その結果何らかの影響がパフォーマンスに現れる。これは、異常な場合と考えられるので、パフォーマンスには問題はない。

【0042】このクラス・タイプ検査を多数のメソッド・コールに再利用可能とするように、前述のコードを編成し得ることも考えられる。同じクラスのいくつかのコール・メソッドおよびターゲット・メソッドのクラスが、あるコールから他のコールにサブクラス化できなかった場合、単一のクラス・タイプ検査を行い、この検査をターゲット・メソッド・コール全てで使うことができる。この実現例は、クラス・タイプ検査を実行することに伴うオーバーヘッドを、複数のメソッド・コールに広げ、これによって効率の向上を図る。

【0043】更にインライニングの改善を得る方法は、クラス検査（例えば、上述の例では、CMPおよびBNE命令）を全て除去することである。その結果得られるコードは、ターゲット・メソッドのコールのみを含み、したがって非常に効率的となる。しかしながら、ターゲット・メソッドが既に無効化されている場合、インラインされたコードは正確でない。したがって、これを行う場合、ターゲット・メソッドのクラスが、そのメソッドを無効化したクラスによってサブクラス化されている場合、そのインラインされたコード・セグメントを仮想コール・シーケンスで置換しなければならないことを、JVMは思い出さなければならない。

【0044】仮想コール・シーケンスは、それと置換するいずれのものよりも大きい可能性があり、したがって

これを解決するためには、元のインライン・コードの最初の命令を、仮想コール・シーケンスを実施する命令がある位置にプログラムの実行を転送する命令に変更することができる。その例には、分岐命令、コール命令、ブレークポイント命令等が含まれる。どのようにしてこの特徴を実施するかについての選択は、個々の機械に適した効率的で柔軟性のある実施を与えるのは何かに基づいて行う。

【0045】図3に示す分岐の実施では、インライン・コードの最初の命令を、分岐命令（例えば、BR X。Xは仮想関数コールを実施するコードの位置である）と置換する。後続の実行時に、インライン・コードを実行せず、代わりに位置Xのコードを実行する。仮想関数コールを実施するコードは、インライン・コード・セグメントの終端に戻る分岐命令で終了しなければならない。

【0046】図4に示す「コール」の実施では、インライン・コードの最初の命令は、仮想関数コールを実施する位置Xにおける命令をコールするコール命令と置換する。コールしたシーケンスの終了時に、リターン命令（図4におけるRTN）が、バッチしたコードにおける丁度コール命令の後ろの位置に、命令の実行を戻す。インライン・コードの2番目の命令には、インライン・コード・セグメントの終端を指し示す分岐命令をバッチする。

【0047】図4に示すコールの実施は、コンピュータの仮想メモリ空間（例えば、SPARCアーキテクチャの場合、4ギガバイトのアドレス空間）内のどこにでも、コールした命令シーケンスを置くことができるようにコール命令を実施する。SPARCアーキテクチャ・プロセッサのような、いくつかのプロセッサでは非常に効率的で柔軟性がある。対照的に、典型的な分岐命令は、当該分岐命令周囲の限られたアドレス空間範囲内にある位置に分岐できるに過ぎない。これは、バッチ・コードを実際に実行時にコンパイルするような場合には、重要な利点となる。例えば、図3および図4の位置Xにあるコードを必要とすると判定した後（即ち、インライ

ン・コードの元になったメソッドが無効化された後)に  
 ようやくこれをコンパイルした場合、パッチング・コード  
 を保持するための空メモリ空間が近くにない場合がある。  
 したがって、パッチング・コードは、分岐命令を通じて  
 アクセス可能なアドレス範囲の外側に位置付けなければ  
 ならない場合がある。あるいは、プログラムには、  
 必要に応じてパッチング・コードを保持するために使用  
 可能な空のメモリ空間を満載することができるが、しか  
 しながら、これはメモリ資源の使用法としてはむしろ非  
 効率的であり、プログラムの大型化(bloat)や、命令  
 キャッシュの管理の複雑化を招く虞れがある。

【0048】図5は、オペレーティング・システムのブ  
 レークポイント・ハンドラを用いて、パッチング・コード  
 を実施する実現例を示す。図5に示すように、インライ  
 ン・コードの最初の命令に、実行時に例外を投入する  
 ブレークポイント命令をパッチする。ブレークポイント  
 命令は、ブレークポイント・ハンドラが捕獲し、ブレー  
 クポイント・テーブル601のようなデータ構造を参照  
 する。各ブレークポイント命令「A」毎に、テーブル6  
 01はブレークポイントを処理するための1組の命令  
 「B」をマップする。一方、Bにおける命令は仮想関数  
 コールを実施することができる。尚、このプロシージャ  
 は非常に遅いシーケンスであるが、このケースが稀にし  
 か起こらない用途環境では、正味のパフォーマンス改善  
 が得られる。これらの非効率性は、プロファイルに記録  
 しておき、次回タスクを走らせる際に補正することがで  
 きる。

【0049】この機構を可能にするために、コンパイラ  
 がメソッドを作成するとき、インライン・コード自体を  
 実施するコード断片だけでなく、仮想関数コールを実施  
 するコード断片も作成しなければならない。この仮想関  
 数コール・コード断片は、ブレークポイント・ハンドラ  
 がコールし、セーブしてあるリターン・アドレスを増分  
 し、インラインされたが現在では無効のコードの終点  
 に、コードの実行をジャンプさせ、次いで割り込みシー  
 ケンスからのリターンを行わせなければならない。パッ  
 チするメモリ位置のアドレス、およびコード断片を保持  
 するメモリ位置のアドレスは、インラインするメソッド  
 と関連するデータ構造に置く。このメソッドがサブクラ  
 ス化するときはいつでも、命令データ構造内の命令にパ  
 ッチを行い、ブレークポイント・ハンドラに、その関連  
 付けを通知する。

【0050】本発明によるシステムおよび方法に関する

魅力的な特徴は、コードを一旦生成したなら、再度コン  
 パイルする必要が全くなく、難しいスタック・パッチン  
 グ・プロシージャが不要であることである。コンパイラ  
 の複雑性およびデータ構造のサイズが多少増大するが、  
 コード実行効率の改善は明白である。

【0051】以上、ある程度の特定性をもって本発明を  
 説明し例示したが、本開示は一例として行ったに過ぎ  
 ず、特許請求の範囲における本発明の精神および範囲か  
 ら逸脱することなく、当業者には、その部分の組み合わ  
 せや構成において数多くの変更を行うことが可能である  
 ことは理解されよう。

【図面の簡単な説明】

【図1】本発明による方法およびシステムを実施するネ  
 ットワーク・コンピュータ環境を示す図である。

【図2】本発明のシステムおよび方法による、プログラ  
 ミング環境を示す図である。

【図3】本発明の実現例の一例を示す図である。

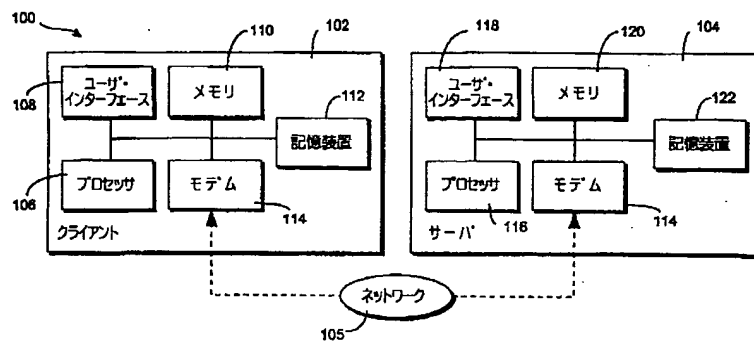
【図4】本発明による別の実施形態を示す図である。

【図5】本発明による更に別の実施形態を示す図であ  
 る。

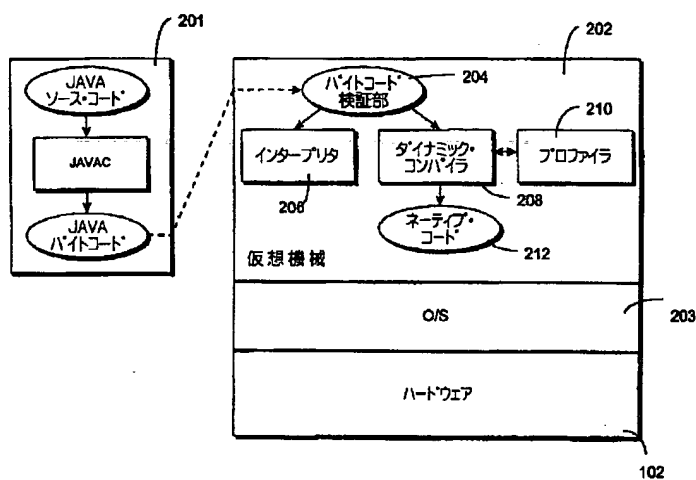
【符号の説明】

100	コンピュータ・システム
102	クライアント・コンピュータ
104	サーバ・コンピュータ
105	ネットワーク
106	プロセッサ・ユニット
108	ユーザ・インターフェース
110	メモリ・ユニット
112	記憶装置
114	モデム
116	プロセッサ
118	ユーザ・インターフェース
120	サーバ・メモリ
122	サーバ記憶装置
201	コンパイル時環境
202	実行時環境
203	オペレーティング・システム
204	バイトコード検証部
206	バイトコード・インタープリタ
208	ダイナミック・コンパイラ
210	プロファイラ
212	コード・キャッシュ

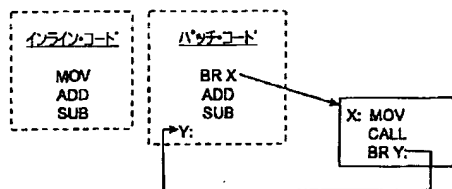
【図1】



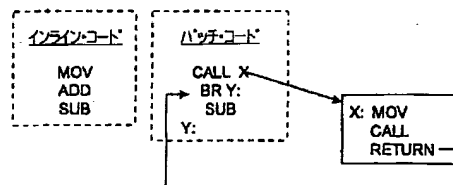
【図2】



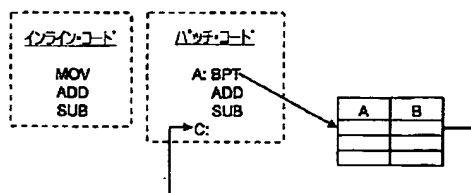
【図3】



【図4】



【図5】



フロントページの続き

(71)出願人 597004720

2550 Garcia Avenue, MS  
PAL1-521, Mountain V  
iew, California 94043-  
1100, United States of  
America